NASA Grant NGR 33-022-125

USING *APL* TO INVESTIGATE SEQUENTIAL MACHINES

Garth Foster

Electrical Engineering Department, Syracuse University

Introduction

Designers of digital systems invariably encounter problems in which the sequence of signal patterns is critical. At these times the creation of the correct finite state sequential machine model is required as a portion of the synthesis procedure. The facility to examine a number of finite state sequential machines, to look at many properties of a single model, or to examine a number of alternate formulations would be a useful tool for computer engineers, logical designers, computer scientists, and teachers. The use of the *APL*\360 Terminal System to provide such a tool allowing the researcher to probe finite state sequential machines is described in this paper. The collection of functions which are collected in a workspace at present have been influenced by some time shared programs written by Thomas F. Piatkowski in BASIC (3 programs) and FORTRAN on the Michigan Terminal System (1 program). These were reported in references [1] and [2] respectively, and following their design objectives allows the interested reader the opportunity to make some comparisons between the functions written in *APL* and good implementations in these more traditional and widely available languages.

Formally a finite state sequential machine, M, is a 6-tuple $M = \langle S, X, Z, \delta, \lambda, s_o \rangle$, where S is the set of states, X is the set of inputs, Z is the set of Outputs; δ is the next state function, mapping ordered pairs of states and inputs into states; $\delta : S \times X \rightarrow S$; and λ is the output function, mapping ordered pairs of states and inputs into outputs, $\lambda : S \times X \rightarrow Z$; and s_o is the initial state, a distinguished member of S. If δ and λ are defined for all members of the Cartesian product $S \times X$, we say the machine is completely specified; otherwise the machine is incomplete. If the machine has an output function λ which is only a function of the state then the model is a Moore machine model; otherwise in the most general form of output function it is a Mealy model.

In our formulation we will represent the set of states by the first *N* integers, ⍳*N* in *ORIGIN* 1 and the set of inputs will be represented by ⍳*P* and the set of outputs by ⍳*Q*. The next state mapping is represented by a variable which is a 2-dimensioned array called ***STATE*** with a row representing the present state and the columns representing the states reached by one of the *P* inputs. In a similar manner we use a matrix, not surprisingly, named ***OUTPUT*** to hold the output mapping. Thus ⍴***OUTPUT*** is ***N***,***P***.

Our model thus deals with ***N***,***P***,***Q*** Mealy machines. By entering 0's appropriately in ***STATE*** or *OUTPUT* we will denote undefined next state or output mappings and thus include incompletely specified machines in the model. How these functions currently relate to incompletely specified machines will be covered later. Suffice it to say that the programs in [1] and [2] deal with complete machines and so some functions described here deal only with complete machines for purposes of comparison.

INITIALIZE and *PRINT*

The function *INITIALIZE* provides a conversational way of setting *N* and *P* and specifying *STATE* and *OUTPUT* with the option of not providing an output function should it not be of importance, as for example, in checking to see if the machine is strongly connected. After specifying the appropriate variables *INITIATIZE* calls *PRINT* to display the machine. ***PRINT*** produces as an explicit result an array of characters giving the mappings contained in ***STATE*** and *OUTPUT* together with some formatting niceties.

At present states and outputs are represented as single characters taken from character vectors *ALPH*1 and *ALPH*2 respectively. Thus for purpose of display "don't care" states and outputs are represented by - and states are represented by letters and outputs by hexadecimal digits. The present restriction in display then is $N \leq 26$ and $Q \leq 16$. Changes in ***PRINT***, *ALPH*1, and *ALPH*2 would remove this restriction, and $P \leq 9$ may be altered by changing ***PRINT***.

These functions are shown in Fig. 1.

In passing we note that separate variables were chosen to store the next state and output mappings rather than storing both mappings as planes of a single array. This is to provide for the future when we will handle Moore type machine models wherein the output is an ***N*** element vector rather than a matrix. At present we may handle this type of machine by the artifice of having all *P* columns of *OUTPUT* be identical.

Simulating a Finite State Machine

It is useful after having specified a particular sequential machine structure to specify a starting state and then upon giving successive inputs to have the corresponding outputs displayed, perhaps with the option of having the current state of the machine displayed or not.

The function *SIMULATE* performs this task, and its display together with a sample of its use are shown in Fig. 2. This function matches, for the most part, the output created by the program SIM in reference [1] (pp. 4-9).

SIM is written in BASIC and contains 72 statements. Among these statements there are 35 PRINT statements, 6 GO TO's, 9 IF's, 5 FOR statements and 5 associated NEXT statements, 7 INPUT, 1 LET statement, 2 DIMension, 1 END and 1 STOP statement. Since the value of variables can only be changed by an INPUT or LET statement, most of the program is clearly printing and sequence control. Moreover, the sequential nature of the problem denies exploitation of the powerful array operators in *APL*. Thus, we should not be surprised when we find that to model the same function in *APL* requires the 17 statement function *SIMULATE*, the 22 statement *INITIALIZE* and the 8 statement function *PRINT*, for a total of 47 reasonably straight forward statements. Still several advantages other than a slight gain in brevity are obtained in the *APL* approach; these are: 1) Modularity. The functions *INITIALIZE* and *PRINT* are usable in their own right and in turn may and will be called by other functions; other programs in [1], for example, duplicate the equivalent of *INITIALIZE* (using 35-38 BASIC statements) at the beginning of each program. 2) Flexibility. Dynamic allocation of memory and a greater flexibility and convenience in input format requirements allow the functions to be adapted to a wider variety of machines. 3) Generality. The *APL* functions shown also accommodate the simulation of incompletely specified machines.

Strongly Connected Machines

A sequential machine is termed strongly connected if for any arbitrary pair of states *I*,*J*, there exists some sequence of inputs which takes the machine from *I* as an initial state to *J* as a final state. It is thus sufficient to be able to go from state 1 to any other state and also to make a (multiple step) transition from any state to state 1.

In the implementation under discussion, this task is performed by *STRONG* which is shown in Fig. 3. To model the BASIC program STR of [1] we also show in Fig. 3 the *APL* function *STR* which acts as calling sequence to *STRONG*, *INITIALIZE* and *PRINT*.

We must comment in passing that due to the modular nature of *APL*, *STR* is really not needed; we may initiate the execution of *INITIALIZE* and then *STRONG* directly from the terminal and although these are separate functions, both manipulate common global variables. The convenience of having functions and variables pooled as resources in a workspace to be used whenever needed is a tremendous advantage of the *APL* Terminal System. In short we need only add the 16 statement function *STRONG* to those already described to check whether a machine is strongly connected. Note that most of the work is done in *STRONG*[2] wherein we start with a set of states, *S*, (initially set to 1) that we can reach from state 1 and then find all the states we can reach with one more input, append that to *S*, sort the set of states, and drop out duplicate entries. When that list has no new entries on it we have the list of states reachable from state 1. In a similar fashion *STRONG*[11] builds a list of states from which state 1 may be reached by some input sequence. Before turning our attention to the problem of minimization, it should be noted that an added advantage of *STRONG* over STR written in BASIC in reference [1] is that *STRONG* lists all states which cannot be reached from state 1 as well as all states which cannot reach state 1 rather than just the first state in lexical order which fulfills one of these qualifications as is done in STR.

Minimization of Sequential Machines

The classical concept of minimization of sequential machines is that of removing superfluous states so as to reduce the number of states in the state table. A state is superfluous if it is equivalent to some other state where we say two states are equivalent if the outputs are equal for any input and if the pair of states reached under any input are also equivalent.

This relationship is an equivalence which induces a partition on the states of the machine breaking that set up into a set of sets such that each of these sets or blocks of the partition has no member in common with any other block of the partition and such that any pair of states within a block are indistinguishable from one another in terms of input-output considerations.

Since all states within a block have the same output for each input, each of the blocks serves as a prototype for a state in a minimized machine. This solves the classical minimization problem providing the machine is completely specified. It is this procedure that has been implemented in the BASIC program MIN which also appears in reference [1] (pp. 17-24).

Unfortunately the concept of equivalence no longer holds if the sequential machine is not completely specified. We do not intend this to be either a tutorial or a review of the minimization of incompletely specified sequential machines; the reader can refer to references [3], [4], [5] or any other recent book on switching and automata theory. However, to provide a better understanding of the *APL* functions which follow, some additional definitions and discussion are required.

Minimal Incomplete Machines

When the specification of the machine is not complete, rather than equivalence of states, we say that two states are compatible if for all inputs, the outputs of the states are identical where they are both defined and where for any input the next states reached by the pair of states are compatible where both next states are defined. This relation between pairs of states is reflexive and symmetric but not transitive. States which are compatible by this definition may be grouped into the largest sets such that each pair of states in a set is pairwise compatible and such that no further state may be added to a set without destroying compatibility. The collection of sets is called the maximal compatibles.

A set of sets, $K = \{B_1, B_2, \ldots, B_p\}$, is called a cover for a set, S, of states if

1) $S = \bigcup_{i=1}^{p} B_i$

The cover is a set system if

2) $B_i \subseteq B_j \quad i=j.$

That is, no block may be a subset of any other block. Further, if a set system has the property that for all i,j

3) $B_i \cap B_j = \emptyset$

then the set system is a partition which we have discussed above. Thus, every partition is a set system but the converse is not true.

Relative to sequential machines, a set system (or partition) is closed or has the substitution property if for any $B_i \in K$, we have $\delta(B_i, J) \subseteq B_j$ for any input J and for some block B_j of K.

Now, the maximal compatibles (equivalence classes) arising from the compatibility (equivalence) relations defined on incompletely (completely) specified sequential machines are closed due to the definition of compatibility (equivalence) of states.

Moreover, if one finds the maximal compatibles for a completely specified machine, the compatibility relation becomes an equivalence and the equivalence classes are thus obtained. Thus, the minimization of a completely specified machine is a subproblem of incomplete machine minimization, and the maximal compatibles solve the problem for complete machines.

On the other hand, for incompletely specified machines having the maximal compatibles is not sufficient for minimization. For while the maximal compatibles are closed, they may not be minimal. A closed cover (a set system) may be derived by considering subsets of the blocks of the maximal compatibles, and this serves as a definition of a minimal machine. This problem is beyond the scope of this paper and it is discussed at length in references [3], [4] and [5] for example and an algorithm is given in [6] for a solution to this problem.

These methods have not yet been implemented in this workspace but we mention the problem here to indicate the direction of further development of the workspace. We also seek to indicate that the algorithms by which we have chosen to minimize complete sequential machines are much more general than the approach used in reference [1] and comparisons of size of programs are difficult to make.

APL Functions for Machine Minimization

The task of computing the maximal compatible sets of states is performed by the four functions *COMP*, *EXP*, *MAXC*, and *SUBSET* each of which produces an explicit result. *COMP* uses global variables *STATE* and *OUTPUT* and an ancillary function *COLS* to create a matrix of 2 columns and less than 2!*N* rows which lists all pairs of states which are compatible. A list of state pairs which are output compatible only is produced as a by-product. The results from *COMP* may then be passed to *EXP* which produces a logical matrix which expresses the compatibility relation. This result is then available to *MAXC* which produces the set system of maximal compatibles. The function *SUBSET* is used to remove subsets from the list so as to maintain a set system during the computations of *MAXC*. These four functions are given in Fig. 4. The result of *MAXC* is a representation of a set system.

At this point we are faced with the problem of how the cover produced by *MAXC* is represented. If the result is a partition, then we are guaranteed of having each of the *N* states appear only once in the list. We know that there can be no more than *N* blocks in the partition and each block has no more than *N* members. What is required to efficiently deal with a cover is the concept of arrays of arrays. Certainly an easy way to handle partitions is to use an *N* element vector, *V*, such that *V*[*I*] is an integer giving the block number of state *I*. Thus, 1 2 1 3 2 4 1 represents the partition 1,3,7; 2,5; 4; 6. The expression ⌈/*V* gives the number of blocks in the partition, ⍋*V* orders the partition without giving the locations where the breaks between blocks occur. The size of the block with the greatest number of elements is relatively easy to derive and this method of representation will be used when we discuss partitions with the substitution property below.

The requirement of being able to represent covers which are set systems is still with us however, and we have chosen to have the cover returned as a result by *MAXC* be represented in the following way.

Let *B*1, *B*2, ... *BL* be vectors representing the *L* blocks of the cover K then the result *V* of *MAXC* will be:

```
V←(L+2),0,(+\(⍴B1),(⍴B2),...,⍴BL),B1,B2,...,BL
```

Then the function *OF*, shown in Fig. 5 may be used to pick up the *X*th block of a cover stored in the format of *V* above.

The other functions of Fig. 5 enable us by use of *MINIMIZE* to do more than just perform the same function as is accomplished by the program MIN of reference [1] for complete machines. *MINIMIZE*[1] calculates the cover (a partition). If the given machine was minimal, we go to line [9] and print a message. If the machine may be reduced we list the equivalence classes by means of *LIST* which in turn makes use of *OF* and finally we *REDUCE* the state table and *PRINT* the new machines tables using the global variables *OLDSTATE* and *OLDOUT* to hold a copy of the original machine in case we want to return it.

Partitions with the Substitution Property

The *N* states of a sequential machine may be partitioned in *PART N* different ways, where *PART* is given by:

```
    ∇ Z←PART N;K;T
[1]   Z←,1+K←0
[2]   Z←(((0,⍳T)!T←¯1+⍴Z)+.×Z),Z
[3]   →2×⍳N>K←K+1
[4]   Z←1↑Z
    ∇
```

While the number of partitions grows quite rapidly as *N* increases, the number of partitions having the substitution property is vanishingly small compared to the result of *PART N* and yet these are the partitions which contain the essence of the flow information in a complete sequential machine. Partitions which are closed may be used to decompose the machine into simpler machines in parallel or cascade or to reduce the interdependency among the ⌈2⍟*N* binary variables which encode the *N* states internal to the machine. An extensive discussion of this topic may be found in reference [7].

Reference [2] contains a FORTRAN program written for the Michigan Terminal System which calculates all of the partitions having the substitution property of a sequential machine and provides sufficient information to construct the lattice of closed partitions.

That construction proceeds in the following manner. Consider a pair of distinct states as being grouped together in a block of a partition. Look at the image of this set under all possible inputs in the next state function, *STATE*, and group those states together as required to maintain closure. This may mean creating new blocks or coalescing two or more blocks into a larger block. When a blocking of the states found by iterating in this manner holds, we have a partition having the substitution property.

These partitions are mapped into themselves by the next state function. This is more restrictive than requiring a partition to be mapped into some other partition by the next state function. Such a requirement leads to the idea of partition pairs, also discussed in reference [7], but which is beyond the present scope of discussion.

By examining all possible 2-state generators in the manner described above we find all possible partitions with the substitution property which are lattice atoms for the machine in question. Also, some, but not necessarily all, closed partitions are derived; however we may derive the missing partitions by taking the *SUM* of presently derived partitions with the substitution property since that property is preserved when we take the sum or intersection of partitions which are themselves closed.

However, before proceeding with the derivation of additional partition we: 1) *NORMALIZE* the generator partitions to place them in the form previously discussed. This makes it easier to compare two partitions for containment or equality as well as conveniently count the number of blocks. 2) *COMPRESS* the list of partitions removing duplicates.

The list of generator partitions is then sorted in decreasing sequence by the number of blocks in the partition for case of search for additional closed partitions.

The primary function *SP* is shown together with its ancillary functions in Fig. 6. The FORTRAN program which they replace was given in reference [2]. That program is 346 statements long, and it consists of a main program, 4 subroutines and one logical function. All told there are 35 FORMAT statements, 5 READS, and 36 WRITES to provide the necessary I/O. There are 89 assignment statements used in expressing the algorithm. Iteration is provided by the use of 48 DO loops with 31 CONTINUE statements. Sequencing of statements is modified by 20 arithmetic IF statements, 26 logical IF statements (some of these provided conditional assignment) and 13 GO TO's. There are 9 CALLs to subroutines and the function calls are generally imbedded in logical IF statements. There are 7 DIMENSION statements and 27 other declarations such as subroutine headers, variable type declarations, RETURNs, ENDs, STOP, or DATA statements. The function *SP* does the essential work of finding the partitions with the substitution property although the input/output format are slightly altered from the program SP in reference [2].

Even with the straightforward modeling of then original algorithms, compactness has been achieved with *APL*. This leads to a representation of the algorithm which is easier to follow and which is to some extent self-documenting by virtue of modularity and other system considerations which tend to eliminate declarations, odd naming conventions, and many of the explicit looping mechanisms dealt with by DO loop structures.

Conclusions

A finite state sequential machine has a matrix-like formulation, but both because of the sequential nature of the device as well as the types of problems associated with sequential machines, matrix formulations of the problem algorithms are harder to come by. Even with the traditional approaches to a problem *APL* provides a more compact representation of the algorithm. In many respects the *APL* formulation embodies a greater degree of clarity which derives from the concise nature of the language.

Rather than compare sizes of a single function or program it is more relevant to consider the modularity inherent in *APL* which gives rise to considerable gains in convenience in creating and using the entire applications package. Here *APL* comes out way ahead. Also, in many applications of the sort that we have discussed where there is a great deal of interaction between the user and the model that he is building the open ended nature of the *APL* Terminal System is an asset. It must also be noted that with a great deal of interaction involved, the interpretive nature of *APL* becomes less of a burden because: 1) most of the time is the person's thinking and reaction time, 2) a model of some sort is generally being developed and/or used and it is the person's time which is to be optimized and 3) in modeling the structure of the model is usually subject to change and that requirement is easily satisfied in an interpretive mode.

References

[1] Piatkowski, Thomas F.: *Computer Programs Dealing with Finite-State Machines: Part I,* Technical Report, Department of Electrical Engineering, University of Michigan, Ann Arbor, May, 1967 (AD-657 999).

[2] Piatkowski, Thomas F.: *Computer Programs Dealing with Finite-State Machines: Part II,* Technical Report, Department of Electrical Engineering, University of Michigan, Ann Arbor, July, 1967 (AD-658 001).

[3] Prather, Ronald E.: Introduction to Switching Theory: A Mathematical Approach, Allyn and Bacon, Boston, 1967.

[4] Hill, Frederick J. and Gerald R. Peterson: Introduction to Switching Theory and Logical Design, John Wiley and Sons, New York, 1968.

[5] Kohavi, Zvi: Switching and Finite Automata, McGraw-Hill, New York, 1970.

[6] Prather, Ronald E.: *Minimal Solutions of Paull-Unger Problems,* Mathematical Systems Theory, vol. 3, (1969), 1, pp. 76-85.

[7] Hartmanis, J. and R. E. Stearns: Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Englewood Cliffs, N. J., 1966.

```
      INITIALIZE
NUMBER OF STATES.N
⎕:
      6
NUMBER OF INPUTS, P
⎕:
      2
ENTER ROWS OF THE STATE TABLE AS RE
      QUESTED
1
⎕:
      5 4
2
⎕:
      6 4
3
⎕:
      5 2
4
⎕:
      6 2
5
⎕:
      3 6
6
⎕:
      2 3
OUTPUT TABLE REQUIRED? (YES, NO)
YES
ENTER ROWS OF THE OUTPUT TABLE AS R
      EQUESTED
1
⎕:
      1 2
2
⎕:
      1 1
3
⎕:
      1 2
4
⎕:
      1 1
5
⎕:
      1 2
6
⎕:
      1 1

  |1    2
-------------
A |E,0  D,1
B |F,0  D,0
C |E,0  B,1
D |F,0  B,0
E |C,0  F,1
F |B,0  C,0
```

```
        ∇INITIALIZE[⎕]∇

      ∇ INITIALIZE;K;T;SW
[1]     'NUMBER OF STATES.N'
[2]     N←⎕
[3]     'NUMBER OF INPUTS, P'
[4]     P←⎕
[5]     STATE←⍳SW←0
[6]     K←1
[7]     'ENTER ROWS OF THE ',((
        5+¯11×SW)↑'STATEOUTPUT'),' T
        ABLE AS REQUESTED'
[8]     K
[9]     →11×⍳P=⍴T←,⎕
[10]    →8,⍴⎕←'SIZE ERROR RE-ENTER R
        OW'
[11]    STATE←STATE,T
[12]    →8×⍳N≥K←K+1
[13]    →16×⍳SW
[14]    'OUTPUT TABLE REQUIRED? (YES
        , NO)'
[15]    →17×⍳'N'∊⍞
[16]    →6×⍳0≠SW←~SW
[17]    →20×⍳(⍴STATE)≠N×P
[18]    OUT←(N,P)⍴0
[19]    →21
[20]    OUT←(N,P)⍴(N×P)↓STATE
[21]    STATE←(N,P)⍴STATE
[22]    PRINT
      ∇

        ∇PRINT[⎕]∇

      ∇ T←PRINT;Q;IN;IP
[1]     T←((5×P)⍴ 1 0 0 0 0)\ALPH1[1
        +STATE]
[2]     T[;2+Q←5×¯1+IP←⍳P]←','
[3]     T[;3+Q]←ALPH2[1+OUT]
[4]     T←(- 2 3 +⍴T)↑T
[5]     T[1;¯1+5×IP]←ALPH2[
        2+IP]
[6]     T[2+IN;1]←ALPH1[1+IN←⍳N]
[7]     T[;3]←'|'
[8]     T[2;]←'-'
      ∇
```

Fig. 1. *INITIALIZE*, *PRINT* and Their Use.

```
      ∇SIMULATE[⎕]∇

    ∇ SIMULATE;S;ST;X;B;I;O
[1]  NEW∆STATE:O←11⍴B←7⍴' '
[2]   'INITIAL STATE?'
[3]   ST←ALPH1[1+S←⎕],5⍴' '
[4]   'STATE SUPPRESS? (YES,NO)  '
[5]   X←'N'∊⍞
[6]   'OUTPUT   ',(2 5 ⍴'     STAT
      E')[1+X;],' INPUT'
[7]  INPUT:I←⍞,0⍴⍞←O,(X/ST),(~X)/B
[8]   I←'123456789'⍳¯1↑I
[9]   →DECIDE×⍳~I∊⍳P
[10]  O←ALPH2[1+,OUT[S;I]],
      10⍴' '
[11]  ST←ALPH1[1+S←,STATE[S;I]],
      5⍴' '
[12]  →INPUT
[13] DECIDE:'ILLEGAL INPUT  (QUIT,
       INPUT, NEW∆MACHINE, NEW∆STA
      TE)'
[14]  →⎕
[15] QUIT:→
[16] NEW∆MACHINE:INITIALIZE
[17]  →1
    ∇
```

```
      SIMULATE
INITIAL STATE?
⎕:
      1
STATE SUPPRESS? (YES,NO)
NO
OUTPUT   STATE INPUT
           A     1
0          E     2
1          F     1
0          B     2
0          D     0
ILLEGAL INPUT  (QUIT, INPUT, NEW∆MA
      CHINE, NEW∆STATE)
⎕:
      INPUT
0          D     1
0          F     8
ILLEGAL INPUT  (QUIT, INPUT, NEW∆MA
      CHINE, NEW∆STATE)
⎕:
      NEW∆STATE
INITIAL STATE?
⎕:
      1
STATE SUPPRESS? (YES,NO)
YES
OUTPUT           INPUT
                   1
0                  2
1                  1
0                  2
0                  0
ILLEGAL INPUT  (QUIT, INPUT, NEW∆MA
      CHINE, NEW∆STATE)
⎕:
      QUIT
```

Fig. 2. *STIMULATE* and Its Use.

```
      ∇STRONG[⎕]∇

    ∇ STRONG;S;T;R;B;IN
[1]   B←S←,1
[2]   T←(1,¯1↓T≠1⌽T)/T←T[⍋T←S,,
      STATE[S;]]
[3]   →5×⍳∧/T∊S
[4]   →2,S←T
[5]   T←(~T∊S)/T←IN←⍳N
[6]   →(10×⍳R)⌊⍴⎕←(R←0=⍴T)/'ALL ST
      ATES ACCESSIBLE FROM STATE 1
      .'
[7]   'MACHINE IS NOT STRONGLY CON
      NECTED.'
[8]   'CAN NOT REACH STATE',((1<⍴T
      )/'S'),' ';T;' FROM STATE',(
      (1<⍴B)/'S'),' ';B;'.'
[9]   →0
[10]  B←(T←S←,1)↓IN
[11]  R←(1,¯1↓R≠1⌽R)/R←R[⍋R←S,(∨/
      STATE[B;]∊S)/B]
[12]  →14×⍳∧/R∊S
[13]  →11,S←R
[14]  B←(~IN∊S)/IN
[15]  →7×⍳0≠⍴B
[16]  'MACHINE IS STRONGLY CONNECT
      ED.'
    ∇

      ∇STR[⎕]∇

    ∇ STR
[1]   INITIALIZE
[2]   STRONG
[3]   'NEW MACHINE?  (YES, NO)'
[4]   →1×'Y'∊⍞
    ∇
```

```
      STRONG
ALL STATES ACCESSIBLE FROM STATE 1.
MACHINE IS NOT STRONGLY CONNECTED.
CAN NOT REACH STATE 1 FROM STATES
      2  3  4  5  6.
```

Fig. 3. *STRONG*, *STR* and the Use of *STRONG*.

```
        ∇COMP[⎕]∇

      ∇ CSP←COMP;I;J;IJ;S1;S2;S3;S4;
        T
[1]     COLS
[2]     OC←∧/(OUT[I;]=0)∨(OUT[J;]=0)
        ∨(OUT[I;]=OUT[J;])
[3]     CSP←OC/[1] IJ
[4]     →0×⍳~∨/OC
[5]     S1←((N+1),1)+.×S1←(
        2,⍴S2)⍴((2×⍴S2)⍴S2≠0)×(S2←S3
        ⌊S4),(S3←,STATE[CSP[;1];])⌈
        S4←,STATE[CSP[;2];]
[6]     T←∧/S0←((⍴CSP)[1],P)⍴S1∊0,((
        (N+1),1)+.×⍉CSP),((N+1),1)+.
        ×(2,N)⍴⍳N
[7]     →0×⍳(⍴T)=+/T
[8]     CSP←T/[1] CSP
[9]     →5
      ∇

        ∇COLS[⎕]∇

      ∇ COLS;TM;CC
[1]     IJ←⍉(2,
        0.5×⍴IJ)⍴IJ←(I←TM/,⍉CC),J←(
        TM←,IN∘.<IN)/,CC←(N,N)⍴IN←⍳N
      ∇

        ∇EXP[⎕]∇

      ∇ RE←EXP PR;K
[1]     RE←(⍳N)∘.=⍳N
[2]     →6×⍳0=⍴,PR
[3]     K←1
[4]     RE[PR[K;1];PR[K;2]]←1
[5]     →4×⍳(⍴PR)[1]≥K←K+1
[6]     RE←RE∨⍉RE
      ∇

        ∇MAXC[⎕]∇

      ∇ R←MAXC CS;I;S;IR;J;C;T;A1;A2
        ;SW1;SW2;Q
[1]     →0×⍳≠/⍴CS
[2]     R←Q←⍳S←(I←1)↑⍴CS
[3]     IR←0,S
[4]     →19×⍳∧/0≠CS[;I]⌈I≥⍳S
[5]     J←1
[6]     T←IR[J]↓IR[J+1]↑R
[7]     →18×⍳~I∊T
[8]     →18×⍳~1∊((~I↑CS[;I])/I↑Q)∊T
[9]     A1←SUBSET(T∊CS[;I]/Q)/T
[10]    A2←SUBSET(I≠T)/T
[11]    SW1←~∧/A1∊A2
[12]    SW2←~∧/A2∊A1
[13]    A1←SW1/A1
[14]    A2←(SW2∨~SW1)/A2
[15]    R←(IR[J]↑R),A1,A2,IR[J+1]↓R
[16]    IR←(J↑IR),((0≠⍴A1)/IR[J]+⍴A1
        ),((0≠⍴A2)/IR[J]+(⍴A1)+⍴A2),
        ((⍴A1)+(⍴A2)+IR[J])+((J+1)↓
        IR)-IR[J+1]
[17]    J←J+1
[18]    →6×⍳(¯1+⍴IR)≥J←J+1
[19]    →4×⍳S>I←I+1
[20]    R←(1+⍴IR),IR,R
      ∇

        ∇SUBSET[⎕]∇

      ∇ Q←SUBSET X;K
[1]     K←1
[2]     Q←X
[3]     →5×⍳K=J
[4]     Q←(~∧/X∊IR[K]↓IR[K+1]↑R)/X
[5]     →0×⍳(K≥¯1+⍴IR)∨0=⍴Q
[6]     →3,K←K+1
      ∇
```

Fig. 4. Functions to Find Maximal Compatibles.

```
        ∇MINIMIZE[⎕]∇

      ∇ MINIMIZE;V
[1]     V←MAXC EXP COMP
[2]     →END×⍳N=V[1]-2
[3]     'EQUIVALENCE CLASSES'
[4]     LIST V
[5]     'THE REDUCED MACHINE:'
[6]     REDUCE V
[7]     PRINT
[8]     →0
[9]   END:'THE GIVEN MACHINE IS MIN
        IMAL'
      ∇

        ∇LIST[⎕]∇

      ∇ LIST V;K;Q
[1]     Q←V[K←1]-2
[2]     K;' *** ';K OF V
[3]     →2×Q≥K←K+1
      ∇

        ∇OF[⎕]∇

      ∇ R←I OF V;J;T
[1]     →2×(I>0)∧I≤V[1]-2
[2]     J←1↓V[1]↑V
[3]     R←J[I]↓J[I+1]↑V[1]↓V
      ∇

        ∇REDUCE[⎕]∇

      ∇ REDUCE V;Q;K;R
[1]     OLDSTATE←STATE
[2]     OLDOUT←OUT
[3]     OUT←STATE←((N←V[1]-
        2),P)⍴0
[4]     K←1
[5]     R←K OF V
[6]     STATE[K;]←⌈⌿OLDSTATE[R;]
[7]     OUT[K;]←⌈⌿OLDOUT[R;]
[8]     →5×⍳N≥K←K+1
[9]     Q←(V[1]↓V)⍳STATE
[10]    K←2↓¯1⌽V[1]↑V
[11]    STATE←+/Q∘.>K
      ∇


        PRINT

  |1     2
-------------
A |E,0   D,1
B |F,0   D,0
C |E,0   B,1
D |F,0   B,0
E |C,0   F,1
F |B,0   C,0


        MINIMIZE
EQUIVALENCE CLASSES
1 *** 1  3
2 *** 2  4
3 *** 5
4 *** 6
THE REDUCED MACHINE:

  |1     2
-------------
A |C,0   B,1
B |D,0   B,0
C |A,0   D,1
D |B,0   A,0
```

Fig. 5. Functions to Minimize Complete Machines and Their Use.

```
       ∇SP[⎕]∇

     ∇ SP;IJ;I;J;TM;CC;G2;N2;K;B;T;
       L;Q;SQ
[1]    COLS
[2]    G2←(N2,N)↑((N2←2!N),1)ρ0
[3]    K←1
[4]   L1:G2[K;IJ[K;]]←1
[5]    →L1×ιN2≥K←K+1
[6]    K←1
[7]   L2:B←1
[8]   L3:T←STATE[(G2[K;]=B)/IN;]
[9]    L←1
[10]  L4:→L5×ι∨/0≠Q←G2[K;T[;L]]
[11]   →L6,G2[K;T[;L]]←1+⌈/G2[K;]
[12]  L5:→L6×ι((ρQ)≠ρSQ)∧∧/SQ∊1↑SQ←
       (Q≠0)/Q
[13]  L7:G2[K;((Q=0)/T[;L]),(G2[K;]
       ∊SQ)/IN]←B←⌊/SQ
[14]   G2[K;]←G2[K;]-Q\+/((Q←G2[K;]
       ≠0)/G2[K;])∘.>(SQ≠B)/SQ
[15]   →L2
[16]  L6:→L4×ιP≥L←L+1
[17]   →L3×ι(⌈/G2[K;])≥B←B+1
[18]   →L2×ιN2≥K←K+1
[19]   K←1
[20]  L8:G2[K;]←NORMALIZE G2[K;]
[21]   →L8×ιN2≥K←K+1
[22]   G2←G2[⍒⌈/G2;]
[23]   B←G2∧.=⍉G2
[24]   COMPRESS
[25]   PP←ιN
[26]   LEVEL←L←0
[27]  L10:Q←(SQ←0=∨⌿B)/ι1↑ρB←ORDER
       G2
[28]   PP←PP,,SQ⌿G2
[29]   LEVEL←LEVEL,(+/SQ)ρL←L+1
[30]   →L14×ι1=ρQ
[31]   I←1
[32]  L11:J←I+1
[33]  L12:→L13×ι∨/G2∧.=T←G2[Q[I];]
       SUM G2[Q[J];]
[34]   G2←(1 0 +ρG2)ρ(,G2),T
[35]  L13:→L12×ι(ρQ)≥J←J+1
[36]   →L11×ι(ρQ)>I←I+1
[37]  L14:G2←(~(ι1↑ρG2)∊Q)⌿G2
[38]   →L10×ι0<×/ρG2
[39]   PP←(((ρPP)÷N),N)ρPP
[40]   K←0
[41]  L15:K;'      ';LEVEL[1+K];'
        ';PRT 1+K
[42]   →L15×(1↑ρPP)>K←K+1
     ∇


       ∇SUM[⎕]∇

     ∇ R←I SUM J;K;B;C;IN
[1]    →0×ι(ρI←,I)≠ρJ←,J
[2]    IN←ιρR←(ρI)ρ0
[3]    K←1
[4]   S1:B←((I∊I[K])/IN) U(J∊J[K])/
       IN
[5]   S2:C←B U((I∊I[B])/IN) U(J∊J[B
       ])/IN
[6]    →S3×ι(∧/C∊B)∧∧/B∊C
[7]    →S2,B←C
[8]   S3:R[B]←K
[9]    →S1×ι(ρR)≥K←Rι0
[10]   R←NORMALIZE R
     ∇
```

```
       ∇COLS[⎕]∇
     ∇ COLS;TM;CC
[1]    IJ←⍉(2,
       0.5×ρIJ)ρIJ←(I←TM/,⍉CC),J←(
       TM←,IN∘.<IN)/,CC←(N,N)ρIN←ιN
     ∇

       ∇NORMALIZE[⎕]∇

     ∇ S←NORMALIZE V;K;P;Q;T;IN
[1]    S←(ρV)ρK←1
[2]    P←1↑Q←IN←ιρV
[3]    S[T←(V∊V[1↑Q])/IN]←P
[4]    P←P+1
[5]    Q←(~Q∊T)/Q
[6]    →3×ι0<ρQ
     ∇

       ∇ORDER[⎕]∇

     ∇ P←ORDER Y;I;J
[1]    P←ι0
[2]    J←(I←1)↑ρY
[3]    P←P,Y COVER I
[4]    →3×ιJ≥I←I+1
[5]    P←(2ρJ)ρP
     ∇

       ∇COVER[⎕]∇

     ∇ S←X COVER I;R;T;Q;K
[1]    R←⌈/X[I;]
[2]    S←I≠ι(K←1)↑ρX
[3]    →5×ι1=ρT←(X[I;]≠K)/ι
       ¯1↑ρX
[4]    S←S∧∧/Q≠1⌽Q←X[;T]
[5]    →3×R≥K←K+1
     ∇

       ∇COMPRESS[⎕]∇

     ∇ COMPRESS;T
[1]    T←ι0
[2]    K←1
[3]    Q←ι1↑ρB
[4]    T←T,G2[B[K;]ι1;]
[5]    K←K++/B[K;]
[6]    →4×ιK≤2!N
[7]    G2←(((ρT)÷N),N)ρT
     ∇

       ∇PRT[⎕]∇

     ∇ Z←PRT K;A;B;C;IN
[1]    C←1↑IN←ιN
[2]    Z←ι0
[3]    B←⌈/PP[K;]
[4]    Z←Z,'(',((1,(2×¯1+ρA)ρ 0 1)\
       A←ALPH1[1+(PP[K;]=C)/IN]),')
       ;'
[5]    →4×ιB≥C←C+1
[6]    Z←¯1↓Z
     ∇

       ∇U[⎕]∇

     ∇ Z←X U Y
[1]    Z←Z[⍋Z←Y,(~X∊Y)/X]
     ∇
```

Fig. 6. *SP* and Associated Functions.

This work was done in conjunction with Air Force Contract F30602-70-C-0190 and NASA Grant NGR 33-022-125.

NASA Grant NGR 33-022-125

REPRINTED FROM "TECHNICAL APPLICATIONS PAPERS - NEREM 70"
70 C 63 NEREM COPYRIGHT 1970

USING *APL* TO INVESTIGATE SEQUENTIAL MACHINES

Garth Foster

Electrical Engineering Department, Syracuse University

Introduction

Designers of digital systems invariably encounter problems in which the sequence of signal patterns is critical. At these times the creation of the correct finite state sequential machine model is required as a portion of the synthesis procedure. The facility to examine a number of finite state sequential machines, to look at many properties of a single model, or to examine a number of alternate formulations would be a useful tool for computer engineers, logical designers, computer scientists, and teachers. The use of the *APL*\360 Terminal System to provide such a tool allowing the researcher to probe finite state sequential machines is described in this paper. The collection of functions which are collected in a workspace at present have been influenced by some time shared programs written by Thomas F. Piatkowski in BASIC (3 programs) and FORTRAN on the Michigan Terminal System (1 program). These were reported in references [1] and [2] respectively, and following their design objectives allows the interested reader the opportunity to make some comparisons between the functions written in *APL* and good implementations in these more traditional and widely available languages.

Formally a finite state sequential machine, M, is a 6-tuple $M = \langle S, X, Z, \delta, \lambda, s_o \rangle$, where S is the set of states, X is the set of inputs, Z is the set of Outputs; δ is the next state function, mapping ordered pairs of states and inputs into states; $\delta: S \times X \rightarrow S$; and λ is the output function, mapping ordered pairs of states and inputs into outputs, $\lambda: S \times X \rightarrow Z$; and s_o is the initial state, a distinguished member of S. If δ and λ are defined for all members of the Cartesian product $S \times X$, we say the machine is completely specified; otherwise the machine is incomplete. If the machine has an output function λ which is only a function of the state then the model is a Moore machine model; otherwise in the most general form of output function it is a Mealy model.

In our formulation we will represent the set of states by the first *N* integers, ⍳*N* in *ORIGIN* 1 and the set of inputs will be represented by ⍳*P* and the set of outputs by ⍳*Q*. The next state mapping is represented by a variable which is a 2-dimensioned array called ***STATE*** with a row representing the present state and the columns representing the states reached by one of the *P* inputs. In a similar manner we use a matrix, not surprisingly, named ***OUTPUT*** to hold the output mapping. Thus ⍴*OUTPUT* is ***N*,*P***.

Our model thus deals with ***N*,*P*,*Q*** Mealy machines. By entering 0's appropriately in ***STATE*** or *OUTPUT* we will denote undefined next state or output mappings and thus include incompletely specified machines in the model. How these functions currently relate to incompletely specified machines will be covered later. Suffice it to say that the programs in [1] and [2] deal with complete machines and so some functions described here deal only with complete machines for purposes of comparison.

INITIALIZE and *PRINT*

The function *INITIALIZE* provides a conversational way of setting ***N*** and ***P*** and specifying *STATE* and *OUTPUT* with the option of not providing an output function should it not be of importance, as for example, in checking to see if the machine is strongly connected. After specifying the appropriate variables *INITIATIZE* calls *PRINT* to display the machine. ***PRINT*** produces as an explicit result an array of characters giving the mappings contained in ***STATE*** and *OUTPUT* together with some formatting niceties.

At present states and outputs are represented as single characters taken from character vectors *ALPH*1 and *ALPH*2 respectively. Thus for purpose of display "don't care" states and outputs are represented by - and states are represented by letters and outputs by hexadecimal digits. The present restriction in display then is ***N*** ≤ 26 and ***Q*** ≤ 16. Changes in ***PRINT***, *ALPH*1, and *ALPH*2 would remove this restriction, and *P*≤9 may be altered by changing ***PRINT***.

These functions are shown in Fig. 1.

In passing we note that separate variables were chosen to store the next state and output mappings rather than storing both mappings as planes of a single array. This is to provide for the future when we will handle Moore type machine models wherein the output is an ***N*** element vector rather than a matrix. At present we may handle this type of machine by the artifice of having all *P* columns of *OUTPUT* be identical.

Simulating a Finite State Machine

It is useful after having specified a particular sequential machine structure to specify a starting state and then upon giving successive inputs to have the corresponding outputs displayed, perhaps with the option of having the current state of the machine displayed or not.

The function *SIMULATE* performs this task, and its display together with a sample of its use are shown in Fig. 2. This function matches, for the most part, the output created by the program SIM in reference [1] (pp. 4-9).

SIM is written in BASIC and contains 72 statements. Among these statements there are 35 PRINT statements, 6 GO TO's, 9 IF's, 5 FOR statements and 5 associated NEXT statements, 7 INPUT, 1 LET statement, 2 DIMension, 1 END and 1 STOP statement. Since the value of variables can only be changed by an INPUT or LET statement, most of the program is clearly printing and sequence control. Moreover, the sequential nature of the problem denies exploitation of the powerful array operators in *APL*. Thus, we should not be surprised when we find that to model the same function in *APL* requires the 17 statement function *SIMULATE*, the 22 statement *INITIALIZE* and the 8 statement function *PRINT*, for a total of 47 reasonably straight forward statements. Still several advantages other than a slight gain in brevity are obtained in the *APL* approach; these are: 1) Modularity. The functions *INITIALIZE* and *PRINT* are usable in their own right and in turn may and will be called by other functions; other programs in [1], for example, duplicate the equivalent of *INITIALIZE* (using 35-38 BASIC statements) at the beginning of each program. 2) Flexibility. Dynamic allocation of memory and a greater flexibility and convenience in input format requirements allow the functions to be adapted to a wider variety of machines. 3) Generality. The *APL* functions shown also accommodate the simulation of incompletely specified machines.

Strongly Connected Machines

A sequential machine is termed strongly connected if for any arbitrary pair of states *I*,*J*, there exists some sequence of inputs which takes the machine from *I* as an initial state to *J* as a final state. It is thus sufficient to be able to go from state 1 to any other state and also to make a (multiple step) transition from any state to state 1.

In the implementation under discussion, this task is performed by *STRONG* which is shown in Fig. 3. To model the BASIC program STR of [1] we also show in Fig. 3 the *APL* function *STR* which acts as calling sequence to *STRONG*, *INITIALIZE* and *PRINT*.

We must comment in passing that due to the modular nature of *APL*, *STR* is really not needed; we may initiate the execution of *INITIALIZE* and then *STRONG* directly from the terminal and although these are separate functions, both manipulate common global variables. The convenience of having functions and variables pooled as resources in a workspace to be used whenever needed is a tremendous advantage of the *APL* Terminal System. In short we need only add the 16 statement function *STRONG* to those already described to check whether a machine is strongly connected. Note that most of the work is done in *STRONG*[2] wherein we start with a set of states, *S*, (initially set to 1) that we can reach from state 1 and then find all the states we can reach with one more input, append that to *S*, sort the set of states, and drop out duplicate entries. When that list has no new entries on it we have the list of states reachable from state 1. In a similar fashion *STRONG*[11] builds a list of states from which state 1 may be reached by some input sequence. Before turning our attention to the problem of minimization, it should be noted that an added advantage of *STRONG* over STR written in BASIC in reference [1] is that *STRONG* lists all states which cannot be reached from state 1 as well as all states which cannot reach state 1 rather than just the first state in lexical order which fulfills one of these qualifications as is done in STR.

Minimization of Sequential Machines

The classical concept of minimization of sequential machines is that of removing superfluous states so as to reduce the number of states in the state table. A state is superfluous if it is equivalent to some other state where we say two states are equivalent if the outputs are equal for any input and if the pair of states reached under any input are also equivalent.

This relationship is an equivalence which induces a partition on the states of the machine breaking that set up into a set of sets such that each of these sets or blocks of the partition has no member in common with any other block of the partition and such that any pair of states within a block are indistinguishable from one another in terms of input-output considerations.

Since all states within a block have the same output for each input, each of the blocks serves as a prototype for a state in a minimized machine. This solves the classical minimization problem providing the machine is completely specified. It is this procedure that has been implemented in the BASIC program MIN which also appears in reference [1] (pp. 17-24).

Unfortunately the concept of equivalence no longer holds if the sequential machine is not completely specified. We do not intend this to be either a tutorial or a review of the minimization of incompletely specified sequential machines; the reader can refer to references [3], [4], [5] or any other recent book on switching and automata theory. However, to provide a better understanding of the *APL* functions which follow, some additional definitions and discussion are required.

Minimal Incomplete Machines

When the specification of the machine is not complete, rather than equivalence of states, we say that two states are compatible if for all inputs, the outputs of the states are identical where they are both defined and where for any input the next states reached by the pair of states are compatible where both next states are defined. This relation between pairs of states is reflexive and symmetric but not transitive. States which are compatible by this definition may be grouped into the largest sets such that each pair of states in a set is pairwise compatible and such that no further state may be added to a set without destroying compatibility. The collection of sets is called the maximal compatibles.

A set of sets, $K = \{B_1, B_2, \ldots, B_p\}$, is called a cover for a set, S, of states if

1) $S = \bigcup_{i=1}^{p} B_i$

The cover is a <u>set system</u> if

2) $B_i \not\subseteq B_j \quad i \neq j.$

That is, no block may be a subset of any other block. Further, if a set system has the property that for all i,j

3) $B_i \cap B_j = \emptyset$

then the set system is a partition which we have discussed above. Thus, every partition is a set system but the converse is not true.

Relative to sequential machines, a set system (or partition) is <u>closed</u> or has the <u>substitution property</u> if for any $B_i \in K$, we have $\delta(B_i, J) \subseteq B_j$ for any input J and for some block B_j of K.

Now, the maximal compatibles (equivalence classes) arising from the compatibility (equivalence) relations defined on incompletely (completely) specified sequential machines are closed due to the definition of compatibility (equivalence) of states.

Moreover, if one finds the maximal compatibles for a completely specified machine, the compatibility relation becomes an equivalence and the equivalence classes are thus obtained. Thus, the minimization of a <u>completely</u> specified machine is a subproblem of incomplete machine minimization, and the maximal compatibles solve the problem for complete machines.

On the other hand, for incompletely specified machines having the maximal compatibles is <u>not</u> sufficient for minimization. For while the maximal compatibles are closed, they may not be minimal. A closed cover (a set system) may be derived by considering subsets of the blocks of the maximal compatibles, and this serves as a definition of a minimal machine. This problem is beyond the scope of this paper and it is discussed at length in references [3], [4] and [5] for example and an algorithm is given in [6] for a solution to this problem.

These methods have not yet been implemented in this workspace but we mention the problem here to indicate the direction of further development of the workspace. We also seek to indicate that the algorithms by which we have chosen to minimize complete sequential machines are much more general than the approach used in reference [1] and comparisons of size of programs are difficult to make.

<u>*APL* Functions for Machine Minimization</u>

The task of computing the maximal compatible sets of states is performed by the four functions *COMP*, *EXP*, *MAXC*, and *SUBSET* each of which produces an explicit result. *COMP* uses global variables *STATE* and *OUTPUT* and an ancillary function *COLS* to create a matrix of 2 columns and less than 2!*N* rows which lists all pairs of states which are compatible. A list of state pairs which are output compatible only is produced as a by-product. The results from *COMP* may then be passed to *EXP* which produces a logical matrix which expresses the compatibility relation. This result is then available to *MAXC* which produces the set system of maximal compatibles. The function *SUBSET* is used to remove subsets from the list so as to maintain a set system during the computations of *MAXC*. These four functions are given in Fig. 4. The result of *MAXC* is a representation of a set system.

At this point we are faced with the problem of how the cover produced by *MAXC* is represented. If the result is a partition, then we are guaranteed of having each of the *N* states appear only once in the list. We know that there can be no more than *N* blocks in the partition and each block has no more than *N* members. What is required to efficiently deal with a cover is the concept of arrays of arrays. Certainly an easy way to handle partitions is to use an *N* element vector, *V*, such that *V*[*I*] is an integer giving the block number of state *I*. Thus, 1 2 1 3 2 4 1 represents the partition 1,3,7; 2,5; 4; 6. The expression ⌈/*V* gives the number of blocks in the partition, ⍋*V* orders the partition without giving the locations where the breaks between blocks occur. The size of the block with the greatest number of elements is relatively easy to derive and this method of representation will be used when we discuss partitions with the substitution property below.

The requirement of being able to represent covers which are set systems is still with us however, and we have chosen to have the cover returned as a result by *MAXC* be represented in the following way.

Let *B*1, *B*2, ... *BL* be vectors representing the *L* blocks of the cover K then the result *V* of *MAXC* will be:

```
V←(L+2),0,(+\(⍴B1),(⍴B2),...,⍴BL),B1,B2,...,BL
```

Then the function *OF*, shown in Fig. 5 may be used to pick up the *X*th block of a cover stored in the format of *V* above.

The other functions of Fig. 5 enable us by use of *MINIMIZE* to do more than just perform the same function as is accomplished by the program MIN of reference [1] for complete machines. *MINIMIZE*[1] calculates the cover (a partition). If the given machine was minimal, we go to line [9] and print a message. If the machine may be reduced we list the equivalence classes by means of *LIST* which in turn makes use of *OF* and finally we *REDUCE* the state table and *PRINT* the new machines tables using the global variables *OLDSTATE* and *OLDOUT* to hold a copy of the original machine in case we want to return it.

<u>Partitions with the Substitution Property</u>

The *N* states of a sequential machine may be partitioned in *PART N* different ways, where *PART* is given by:

```
     ∇ Z←PART N;K;T
[1]    Z←,1↑K←0
[2]    Z←(((0,⍳T)!T←¯1+⍴Z)+.×Z),Z
[3]    →2×⍳N>K←K+1
[4]    Z←1↑Z
     ∇
```

While the number of partitions grows quite rapidly as *N* increases, the number of partitions having the substitution property is vanishingly small compared to the result of *PART N* and yet these are the partitions which contain the essence of the flow information in a complete sequential machine. Partitions which are closed may be used to decompose the machine into simpler machines in parallel or cascade or to reduce the interdependency among the ⌈2⍟*N* binary variables which encode the *N* states internal to the machine. An extensive discussion of this topic may be found in reference [7].

Reference [2] contains a FORTRAN program written for the Michigan Terminal System which calculates all of the partitions having the substitution property of a sequential machine and provides sufficient information to construct the lattice of closed partitions.

That construction proceeds in the following manner. Consider a pair of distinct states as being grouped together in a block of a partition. Look at the image of this set under all possible inputs in the next state function, *STATE*, and group those states together as required to maintain closure. This may mean creating new blocks or coalescing two or more blocks into a larger block. When a blocking of the states found by iterating in this manner holds, we have a partition having the substitution property.

These partitions are mapped into themselves by the next state function. This is more restrictive than requiring a partition to be mapped into some other partition by the next state function. Such a requirement leads to the idea of partition pairs, also discussed in reference [7], but which is beyond the present scope of discussion.

By examining all possible 2-state generators in the manner described above we find all possible partitions with the substitution property which are lattice atoms for the machine in question. Also, some, but not necessarily all, closed partitions are derived; however we may derive the missing partitions by taking the *SUM* of presently derived partitions with the substitution property since that property is preserved when we take the sum or intersection of partitions which are themselves closed.

However, before proceeding with the derivation of additional partition we: 1) *NORMALIZE* the generator partitions to place them in the form previously discussed. This makes it easier to compare two partitions for containment or equality as well as conveniently count the number of blocks. 2) *COMPRESS* the list of partitions removing duplicates.

The list of generator partitions is then sorted in decreasing sequence by the number of blocks in the partition for case of search for additional closed partitions.

The primary function *SP* is shown together with its ancillary functions in Fig. 6. The FORTRAN program which they replace was given in reference [2]. That program is 346 statements long, and it consists of a main program, 4 subroutines and one logical function. All told there are 35 FORMAT statements, 5 READS, and 36 WRITES to provide the necessary I/O. There are 89 assignment statements used in expressing the algorithm. Iteration is provided by the use of 48 DO loops with 31 CONTINUE statements. Sequencing of statements is modified by 20 arithmetic IF statements, 26 logical IF statements (some of these provided conditional assignment) and 13 GO TO's. There are 9 CALLs to subroutines and the function calls are generally imbedded in logical IF statements. There are 7 DIMENSION statements and 27 other declarations such as subroutine headers, variable type declarations, RETURNs, ENDs, STOP, or DATA statements. The function *SP* does the essential work of finding the partitions with the substitution property although the input/output format are slightly altered from the program SP in reference [2].

Even with the straightforward modeling of then original algorithms, compactness has been achieved with *APL*. This leads to a representation of the algorithm which is easier to follow and which is to some extent self-documenting by virtue of modularity and other system considerations which tend to eliminate declarations, odd naming conventions, and many of the explicit looping mechanisms dealt with by DO loop structures.

Conclusions

A finite state sequential machine has a matrix-like formulation, but both because of the sequential nature of the device as well as the types of problems associated with sequential machines, matrix formulations of the problem algorithms are harder to come by. Even with the traditional approaches to a problem *APL* provides a more compact representation of the algorithm. In many respects the *APL* formulation embodies a greater degree of clarity which derives from the concise nature of the language.

Rather than compare sizes of a single function or program it is more relevant to consider the modularity inherent in *APL* which gives rise to considerable gains in convenience in creating and using the entire applications package. Here *APL* comes out way ahead. Also, in many applications of the sort that we have discussed where there is a great deal of interaction between the user and the model that he is building the open ended nature of the *APL* Terminal System is an asset. It must also be noted that with a great deal of interaction involved, the interpretive nature of *APL* becomes less of a burden because: 1) most of the time is the person's thinking and reaction time, 2) a model of some sort is generally being developed and/or used and it is the person's time which is to be optimized and 3) in modeling the structure of the model is usually subject to change and that requirement is easily satisfied in an interpretive mode.

References

[1] Piatkowski, Thomas F.: *Computer Programs Dealing with Finite-State Machines: Part I*, Technical Report, Department of Electrical Engineering, University of Michigan, Ann Arbor, May, 1967 (AD-657 999).

[2] Piatkowski, Thomas F.: *Computer Programs Dealing with Finite-State Machines: Part II*, Technical Report, Department of Electrical Engineering, University of Michigan, Ann Arbor, July, 1967 (AD-658 001).

[3] Prather, Ronald E.: Introduction to Switching Theory: A Mathematical Approach, Allyn and Bacon, Boston, 1967.

[4] Hill, Frederick J. and Gerald R. Peterson: Introduction to Switching Theory and Logical Design, John Wiley and Sons, New York, 1968.

[5] Kohavi, Zvi: Switching and Finite Automata, McGraw-Hill, New York, 1970.

[6] Prather, Ronald E.: *Minimal Solutions of Paull-Unger Problems*, Mathematical Systems Theory, vol. 3, (1969), 1, pp. 76-85.

[7] Hartmanis, J. and R. E. Stearns: Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Englewood Cliffs, N. J., 1966.

```
      INITIALIZE
NUMBER OF STATES,N
⎕:
      6
NUMBER OF INPUTS, P
⎕:
      2
ENTER ROWS OF THE STATE TABLE AS RE
      QUESTED
1
⎕:
      5 4
2
⎕:
      6 4
3
⎕:
      5 2
4
⎕:
      6 2
5
⎕:
      3 6
6
⎕:
      2 3
OUTPUT TABLE REQUIRED? (YES, NO)
YES
ENTER ROWS OF THE OUTPUT TABLE AS R
      EQUESTED
1
⎕:
      1 2
2
⎕:
      1 1
3
⎕:
      1 2
4
⎕:
      1 1
5
⎕:
      1 2
6
⎕:
      1 1

  |1    2
-------------
A |E,0  D,1
B |F,0  D,0
C |E,0  B,1
D |F,0  B,0
E |C,0  F,1
F |B,0  C,0
```

```
        ∇INITIALIZE[⎕]∇


      ∇ INITIALIZE;K;T;SW
[1]     'NUMBER OF STATES,N'
[2]     N←⎕
[3]     'NUMBER OF INPUTS, P'
[4]     P←⎕
[5]     STATE←⍳SW←0
[6]     K←1
[7]     'ENTER ROWS OF THE ',((
        5+¯11×SW)↑'STATEOUTPUT'),' T
        ABLE AS REQUESTED'
[8]     K
[9]     →11×⍳P=⍴T←,⎕
[10]    →8,⍴⎕←'SIZE ERROR RE-ENTER R
        OW'
[11]    STATE←STATE,T
[12]    →8×⍳N≥K←K+1
[13]    →16×⍳SW
[14]    'OUTPUT TABLE REQUIRED? (YES
        , NO)'
[15]    →17×⍳'N'∊⍞
[16]    →6×⍳0≠SW←~SW
[17]    →20×⍳(⍴STATE)≠N×P
[18]    OUT←(N,P)⍴0
[19]    →21
[20]    OUT←(N,P)⍴(N×P)↓STATE
[21]    STATE←(N,P)⍴STATE
[22]    PRINT
      ∇


        ∇PRINT[⎕]∇


      ∇ T←PRINT;Q;IN;IP
[1]     T←((5×P)⍴ 1 0 0 0 0)\ALPH1[1
        +STATE]
[2]     T[;2+Q←5×¯1+IP←⍳P]←','
[3]     T[;3+Q]←ALPH2[1+OUT]
[4]     T←(- 2 3 +⍴T)↑T
[5]     T[1;¯1+5×IP]←ALPH2[
        2+IP]
[6]     T[2+IN;1]←ALPH1[1+IN←⍳N]
[7]     T[;3]←'|'
[8]     T[2;]←'-'
      ∇
```

Fig. 1. *INITIALIZE, PRINT* and Their Use.

```
      ∇SIMULATE[⎕]∇

    ∇ SIMULATE;S;ST;X;B;I;O
[1]  NEW∆STATE:O←11⍴B←7⍴' '
[2]   'INITIAL STATE?'
[3]   ST←ALPH1[1+S←⎕],5⍴' '
[4]   'STATE SUPPRESS? (YES,NO)  '
[5]   X←'N'∊⍞
[6]   'OUTPUT   ',(2 5 ⍴'     STAT
      E')[1+X;],' INPUT'
[7]  INPUT:I←⍞,0⍴⍞←O,(X/ST),(~X)/B
[8]   I←'123456789'⍳¯1↑I
[9]   →DECIDE×⍳~I∊⍳P
[10]  O←ALPH2[1+,OUT[S;I]],
      10⍴' '
[11]  ST←ALPH1[1+S←,STATE[S;I]],
      5⍴' '
[12]  →INPUT
[13] DECIDE:'ILLEGAL INPUT  (QUIT,
       INPUT, NEW∆MACHINE, NEW∆STA
      TE)'
[14]  →⎕
[15] QUIT:→
[16] NEW∆MACHINE:INITIALIZE
[17]  →1
    ∇


      SIMULATE
INITIAL STATE?
⎕:
      1
STATE SUPPRESS? (YES,NO)
NO
OUTPUT   STATE INPUT
           A     1
0          E     2
1          F     1
0          B     2
0          D     0
ILLEGAL INPUT  (QUIT, INPUT, NEW∆MA
      CHINE, NEW∆STATE)
⎕:
      INPUT
0          D     1
0          F     8
ILLEGAL INPUT  (QUIT, INPUT, NEW∆MA
      CHINE, NEW∆STATE)
⎕:
      NEW∆STATE
INITIAL STATE?
⎕:
      1
STATE SUPPRESS? (YES,NO)
YES
OUTPUT         INPUT
                 1
0                2
1                1
0                2
0                0
ILLEGAL INPUT  (QUIT, INPUT, NEW∆MA
      CHINE, NEW∆STATE)
⎕:
      QUIT
```

Fig. 2. *STIMULATE* and Its Use.

```
      ∇STRONG[⎕]∇

    ∇ STRONG;S;T;R;B;IN
[1]   B←S←,1
[2]   T←(1,¯1↓T≠1⌽T)/T←T[⍋T←S,,
      STATE[S;]]
[3]   →5×⍳∧/T∊S
[4]   →2,S←T
[5]   T←(~T∊S)/T←IN←⍳N
[6]   →(10×⍳R)⌊⍴⎕←(R←0=⍴T)/'ALL ST
      ATES ACCESSIBLE FROM STATE 1
      .'
[7]   'MACHINE IS NOT STRONGLY CON
      NECTED.'
[8]   'CAN NOT REACH STATE',((1<⍴T
      )/'S'),' ';T;' FROM STATE',(
      (1<⍴B)/'S'),' ';B;'.'
[9]   →0
[10]  B←(T←S←,1)↓IN
[11]  R←(1,¯1↓R≠1⌽R)/R←R[⍋R←S,(∨/
      STATE[B;]∊S)/B]
[12]  →14×⍳∧/R∊S
[13]  →11,S←R
[14]  B←(~IN∊S)/IN
[15]  →7×⍳0≠⍴B
[16]  'MACHINE IS STRONGLY CONNECT
      ED.'
    ∇

      ∇STR[⎕]∇

    ∇ STR
[1]   INITIALIZE
[2]   STRONG
[3]   'NEW MACHINE?  (YES, NO)'
[4]   →1×'Y'∊⍞
    ∇


      STRONG
ALL STATES ACCESSIBLE FROM STATE 1.
MACHINE IS NOT STRONGLY CONNECTED.
CAN NOT REACH STATE 1 FROM STATES
      2  3  4  5  6.
```

Fig. 3. *STRONG*, *STR* and the Use of *STRONG*.

```
      ∇COMP[⎕]∇

    ∇ CSP←COMP;I;J;IJ;S1;S2;S3;S4;
      T
[1]   COLS
[2]   OC←∧/(OUT[I;]=0)∨(OUT[J;]=0)
      ∨(OUT[I;]=OUT[J;])
[3]   CSP←OC/[1] IJ
[4]   →0×ι~∨/OC
[5]   S1←((N+1),1)+.×S1←(
      2,ρS2)ρ((2×ρS2)ρS2≠0)×(S2←S3
      ⌊S4),(S3←,STATE[CSP[;1];])⌈
      S4←,STATE[CSP[;2];]
[6]   T←∧/S0←((ρCSP)[1],P)ρS1∊0,((
      (N+1),1)+.×⍉CSP),((N+1),1)+.
      ×(2,N)ριN
[7]   →0×ι(ρT)=+/T
[8]   CSP←T/[1] CSP
[9]   →5
    ∇

      ∇COLS[⎕]∇

    ∇ COLS;TM;CC
[1]   IJ←⍉(2,
      0.5×ρIJ)ρIJ←(I←TM/,⍉CC),J←(
      TM←,IN∘.<IN)/,CC←(N,N)ρIN←ιN
    ∇

      ∇EXP[⎕]∇

    ∇ RE←EXP PR;K
[1]   RE←(ιN)∘.=ιN
[2]   →6×ι0=ρ,PR
[3]   K←1
[4]   RE[PR[K;1];PR[K;2]]←1
[5]   →4×ι(ρPR)[1]≥K←K+1
[6]   RE←RE∨⍉RE
    ∇

      ∇MAXC[⎕]∇

    ∇ R←MAXC CS;I;S;IR;J;C;T;A1;A2
      ;SW1;SW2;Q
[1]   →0×ι≠/ρCS
[2]   R←Q←ιS←(I←1)↑ρCS
[3]   IR←0,S
[4]   →19×ι∧/0≠CS[;I]⌈I≥ιS
[5]   J←1
[6]   T←IR[J]↓IR[J+1]↑R
[7]   →18×ι~I∊T
[8]   →18×ι~1∊((~I↓CS[;I])/I↓Q)∊T
[9]   A1←SUBSET(T∊CS[;I]/Q)/T
[10]  A2←SUBSET(I≠T)/T
[11]  SW1←~∧/A1∊A2
[12]  SW2←~∧/A2∊A1
[13]  A1←SW1/A1
[14]  A2←(SW2∨~SW1)/A2
[15]  R←(IR[J]↑R),A1,A2,IR[J+1]↓R
[16]  IR←(J↑IR),((0≠ρA1)/IR[J]+ρA1
      ),((0≠ρA2)/IR[J]+(ρA1)+ρA2),
      ((ρA1)+(ρA2)+IR[J])+((J+1)↓
      IR)-IR[J+1]
[17]  J←J+1
[18]  →6×ι(¯1+ρIR)≥J←J+1
[19]  →4×ιS>I←I+1
[20]  R←(1+ρIR),IR,R
    ∇

      ∇SUBSET[⎕]∇

    ∇ Q←SUBSET X;K
[1]   K←1
[2]   Q←X
[3]   →5×ιK=J
[4]   Q←(~∧/X∊IR[K]↓IR[K+1]↑R)/X
[5]   →0×ι(K≥¯1+ρIR)∨0=ρQ
[6]   →3,K←K+1
    ∇
```

Fig. 4. Functions to Find Maximal Compatibles.

```
      ∇MINIMIZE[⎕]∇

    ∇ MINIMIZE;V
[1]   V←MAXC EXP COMP
[2]   →END×ιN=V[1]-2
[3]   'EQUIVALENCE CLASSES'
[4]   LIST V
[5]   'THE REDUCED MACHINE:'
[6]   REDUCE V
[7]   PRINT
[8]   →0
[9]  END:'THE GIVEN MACHINE IS MIN
      IMAL'
    ∇

      ∇LIST[⎕]∇

    ∇ LIST V;K;Q
[1]   Q←V[K←1]-2
[2]   K;' *** ';K OF V
[3]   →2×Q≥K←K+1
    ∇

      ∇OF[⎕]∇

    ∇ R←I OF V;J;T
[1]   →2×(I>0)∧I≤V[1]-2
[2]   J←1↓V[1]↑V
[3]   R←J[I]↓J[I+1]↑V[1]↓V
    ∇

      ∇REDUCE[⎕]∇

    ∇ REDUCE V;Q;K;R
[1]   OLDSTATE←STATE
[2]   OLDOUT←OUT
[3]   OUT←STATE←((N←V[1]-
      2),P)ρ0
[4]   K←1
[5]   R←K OF V
[6]   STATE[K;]←⌈⌿OLDSTATE[R;]
[7]   OUT[K;]←⌈⌿OLDOUT[R;]
[8]   →5×ιN≥K←K+1
[9]   Q←(V[1]↓V)ιSTATE
[10]  K←2↓¯1⌽V[1]↑V
[11]  STATE←+/Q∘.>K
    ∇


      PRINT

   |1     2
-------------
A  |E,0   D,1
B  |F,0   D,0
C  |E,0   B,1
D  |F,0   B,0
E  |C,0   F,1
F  |B,0   C,0


      MINIMIZE
EQUIVALENCE CLASSES
1 *** 1  3
2 *** 2  4
3 *** 5
4 *** 6
THE REDUCED MACHINE:

   |1     2
-------------
A  |C,0   B,1
B  |D,0   B,0
C  |A,0   D,1
D  |B,0   A,0
```

Fig. 5. Functions to Minimize Complete Machines and Their Use.

```
      ∇SP[⎕]∇

    ∇ SP;IJ;I;J;TM;CC;G2;N2;K;B;T;
      L;Q;SQ
[1]     COLS
[2]     G2←(N2,N)↑((N2←2!N),1)ρ0
[3]     K←1
[4]   L1:G2[K;IJ[K;]]←1
[5]     →L1×ιN2≥K←K+1
[6]     K←1
[7]   L2:B←1
[8]   L3:T←STATE[(G2[K;]=B)/IN;]
[9]     L←1
[10]  L4:→L5×ι∨/0≠Q←G2[K;T[;L]]
[11]    →L6,G2[K;T[;L]]←1+⌈/G2[K;]
[12]  L5:→L6×ι((ρQ)≠ρSQ)∧∧/SQ∊1↑SQ←
        (Q≠0)/Q
[13]  L7:G2[K;((Q=0)/T[;L]),(G2[K;]
        ∊SQ)/IN]←B←⌊/SQ
[14]    G2[K;]←G2[K;]-Q\+/((Q←G2[K;]
        ≠0)/G2[K;])∘.>(SQ≠B)/SQ
[15]    →L2
[16]  L6:→L4×ιP≥L←L+1
[17]    →L3×ι(⌈/G2[K;])≥B←B+1
[18]    →L2×ιN2≥K←K+1
[19]    K←1
[20]  L8:G2[K;]←NORMALIZE G2[K;]
[21]    →L8×ιN2≥K←K+1
[22]    G2←G2[⍒⌈/G2;]
[23]    B←G2∧.=⍉G2
[24]    COMPRESS
[25]    PP←ιN
[26]    LEVEL←L←0
[27]  L10:Q←(SQ←0=∨⌿B)/ι1↑ρB←ORDER
        G2
[28]    PP←PP,,SQ⌿G2
[29]    LEVEL←LEVEL,(+/SQ)ρL←L+1
[30]    →L14×ι1=ρQ
[31]    I←1
[32]  L11:J←I+1
[33]  L12:→L13×ι∨/G2∧.=T←G2[Q[I];]
        SUM G2[Q[J];]
[34]    G2←(1 0 +ρG2)ρ(,G2),T
[35]  L13:→L12×ι(ρQ)≥J←J+1
[36]    →L11×ι(ρQ)>I←I+1
[37]  L14:G2←(~(ι1↑ρG2)∊Q)⌿G2
[38]    →L10×ι0<×/ρG2
[39]    PP←(((ρPP)÷N),N)ρPP
[40]    K←0
[41]  L15:K;'      ';LEVEL[1+K];'
          ';PRT 1+K
[42]    →L15×(1↑ρPP)>K←K+1
    ∇


      ∇SUM[⎕]∇

    ∇ R←I SUM J;K;B;C;IN
[1]     →0×ι(ρI←,I)≠ρJ←,J
[2]     IN←ιρR←(ρI)ρ0
[3]     K←1
[4]   S1:B←((I∊I[K])/IN) U(J∊J[K])/
        IN
[5]   S2:C←B U((I∊I[B])/IN) U(J∊J[B
        ])/IN
[6]     →S3×ι(∧/C∊B)∧∧/B∊C
[7]     →S2,B←C
[8]   S3:R[B]←K
[9]     →S1×ι(ρR)≥K←Rι0
[10]    R←NORMALIZE R
    ∇
```

```
      ∇COLS[⎕]∇
    ∇ COLS;TM;CC
[1]     IJ←⍉(2,
        0.5×ρIJ)ρIJ←(I←TM/,⍉CC),J←(
        TM←,IN∘.<IN)/,CC←(N,N)ρIN←ιN
    ∇

      ∇NORMALIZE[⎕]∇

    ∇ S←NORMALIZE V;K;P;Q;T;IN
[1]     S←(ρV)ρK←1
[2]     P←1↑Q←IN←ιρV
[3]     S[T←(V∊V[1↑Q])/IN]←P
[4]     P←P+1
[5]     Q←(~Q∊T)/Q
[6]     →3×ι0<ρQ
    ∇

      ∇ORDER[⎕]∇

    ∇ P←ORDER Y;I;J
[1]     P←ι0
[2]     J←(I←1)↑ρY
[3]     P←P,Y COVER I
[4]     →3×ιJ≥I←I+1
[5]     P←(2ρJ)ρP
    ∇

      ∇COVER[⎕]∇

    ∇ S←X COVER I;R;T;Q;K
[1]     R←⌈/X[I;]
[2]     S←I≠ι(K←1)↑ρX
[3]     →5×ι1=ρT←(X[I;]≠K)/ι
        ¯1↑ρX
[4]     S←S∧∧/Q=1⌽Q←X[;T]
[5]     →3×R≥K←K+1
    ∇

      ∇COMPRESS[⎕]∇

    ∇ COMPRESS;T
[1]     T←ι0
[2]     K←1
[3]     Q←ι1↑ρB
[4]     T←T,G2[B[K;]ι1;]
[5]     K←K++/B[K;]
[6]     →4×ιK≤2!N
[7]     G2←(((ρT)÷N),N)ρT
    ∇

      ∇PRT[⎕]∇

    ∇ Z←PRT K;A;B;C;IN
[1]     C←1↑IN←ιN
[2]     Z←ι0
[3]     B←⌈/PP[K;]
[4]     Z←Z,'(',((1,(2×¯1+ρA)ρ 0 1)\
        A←ALPH1[1+(PP[K;]≠C)/IN]),')
        ;'
[5]     →4×ιB≥C←C+1
[6]     Z←¯1↓Z
    ∇

      ∇U[⎕]∇

    ∇ Z←X U Y
[1]     Z←Z[⍋Z←Y,(~X∊Y)/X]
    ∇
```

Fig. 6. *SP* and Associated Functions.

This work was done in conjunction with Air Force Contract F30602-70-C-0190 and NASA Grant NGR 33-022-125.